Developer's Guide

Earthdata Search Development Guide

Getting Started

Improving the Guide

This guide is a constant work in progress. If something is missing or confusing, please update the guide with better information.

Prerequisites

To develop Earthdata Search, you will need a computer running Mac OS X 10.8+ and the following:

1. RVM (http://rvm.io/)
2. Homebrew (http://brew.sh/)
3. Qt (`brew install qt`)
4. git (from Xcode developer tools)

Obtaining Code

1. Create an account on the Earthdata Code Collaborative (https://ecc.earthdata.nasa.gov/) and request access to the EDSC project.
2. Once you have access, you can view the code online at https://git.earthdata.nasa.gov/projects/EDSC
3. To clone the repository, run `git clone https://<username>@git.earthdata.nasa.gov/scm/edsc/earthdata-search-client_repo.git`, replacing `<username>` with your ECC username.

Running Code

Start by updating the project:

```
$ cd <project-dir>
    $ git pull
    $ bundle install
```

To run a development instance:

```
$ rake db:migrate
    $ rails s
```

Common Commands

Command Description

`rake -T`

List rake commands available in the project

`rake doc`

Generate documentation. UI documentation is generated under doc/ui/index.html

`rake test`

Run all tests, including unit, functional, integration, and UI tests

`rake jasmine`

Run Jasmine / Coffeescript specs in a browser

`rake spec`

Run RSpec specs

`rake test:csslint`

Run CSSLint on generated documentation

`rake test:deadweight`

Run deadweight to find unexercised CSS in UI documentation

`rails s`

Run the app locally

`rake notes`

View all TODO and FIXME comments

Resources

JIRA Agile Board: https://bugs.earthdata.nasa.gov/secure/RapidBoard.jspa?rapidView=209

JIRA Project: https://bugs.earthdata.nasa.gov/browse/EDSC

Git Stash site: https://git.earthdata.nasa.gov/projects/EDSC

Bamboo Build: https://ci.earthdata.nasa.gov/browse/EDSC-EDSCDB/

Communication

- Stand-ups occur daily at 8:45 AM on the ECHO conference line.
- Sprint planning occurs every second Friday from 10:00AM - 11:00AM over GoToMeeting.
- Retrospectives and sprint overviews take place on the Friday before the start of the sprint.
- Developers should use NASA Jabber accounts for chat and join the Jabber group chat `earthdatadev@conference.im.nasa.gov`. Developers use Messages or Screen Hero for remote screen sharing.

Contact the project lead for up-to-date meeting invitations.

Project Overview

Development Process

Earthdata Search uses an agile development process with two-week sprints. At all times, our master branch maintains a potentially releasable product increment, and we would like to eventually move to a continuous deployment model.

Sprints give us periodic checkpoints for feedback but are orthogonal to our releases.

Meetings

We try very hard to minimize interruptions. As such, our meetings are few and short. We have a combined sprint review and sprint planning before the start of each sprint where stakeholders are invited to see what we've done and give input on where we are going.

We have short daily stand-ups near the start of the work day (8:45 AM) to give status and note any problems.

We have periodic retrospectives, but more importantly we try to maintain direct communication and adjust for problems as they arise.

Issue Types

We use JIRA issue types as follows:

- **Story:** A user-visible requirement which ought to be achievable by a single developer in a couple of days. These are captured in a consistent format, "As a [user type] user, I should <feature description> so that <value to user>". For instance, "As a user, I should be able to log in so that I may access information saved with my account."
- **Epic:** Groups similar stories into less granular pieces of functionality.
- **Bug:** A user-visible implementation problem which causes clearly broken behavior. Examples: a button that does not work, severely degraded performance, a security hole.
- **Improvement:** A user-visible implementation problem which, while not clearly broken, poses usability challenges. These are almost exclusively reserved for interface design issues, since we often have a backlog of features which have been implemented but need UI polish. Improvements are optional and we could ship without them. Examples: a button which looks out of place, text that needs styling.
- **Task:** Necessary work that is not user-visible. Examples: Setting up a CI environment, research to make a recommendation, documentation.
- **New Feature:** Not used.

Estimation

We estimate stories on an exponential scale, 1, 2, 4, or 8 points. These units do not correspond directly to developer-hours.

- **1 Point:** Trivial. Typically a change in configuration only. Example: Sorting a list by a new field when the logic for sorting is already in place.
- **2 Points:** Easy. Changes are straightforward and do not involve complex logic or new components. Example: Allowing the user to delete resources already displayed in a list.
- **4 Points:** Normal. Changes may involve complex logic, a new component, or cross-cutting changes to the codebase. This is the ideal granularity for stories. Example: Creating a new page which presents a list of resources (including appropriate links to that page, and tests for those links and the contents of the page).
- **8 Points:** Hard. Changes involve very difficult logic, extensive testing, or sweeping changes to the codebase. Implementation details may be highly uncertain and risky. Cannot reasonably be broken up into smaller stories. Use this estimation sparingly (only 2% of our v1 stories fell into this category). Example: Drawing ECHO spherical polygons on a Leaflet map.

Note: The following may change as the project transitions to sustaining engineering

Our estimation may differ from typical agile projects in the following ways:

We only assign points to stories. Epics are merely an aggregation of stories. Tasks provide no user-visible value. Bugs and improvements are necessary changes that should have been done as part of the original story. Assigning points to bugs or improvements is essentially double-dipping and ruins velocity metrics.

To see this, imagine a backlog with 100 points worth of stories. Let's say we're assigning points to bugs and have an average of 1 point of bugs for every 2 points of stories. Our velocity is 15 points per sprint. We look at the backlog and think we'll be done in about (100 / 15) = 7 sprints. That's wrong, though. 5 points of our velocity is due to fixing bugs, and the backlog doesn't contain bugs. We are assuming that we will write bug-free code for the next 7 sprints, despite evidence to the contrary.

Now assume we don't assign points to bugs. Our velocity is lower, only 10, because we're spending time fixing bugs we've introduced. It also reflects our defect rate. We are finishing 10 story points in a sprint and however many bugs we've introduced. We look at our backlog and say we'll be done in about (100 / 10) = 10 sprints, because our velocity accurately reflects the time spent fixing bugs we're introducing along the way.

Bugs are tackled immediately. In order for the above justifications to work, we cannot have a huge backlog of bugs. If we've marked something done and it is defective, we fix it. Quality is non-negotiable.

We do not re-estimate stories. The justification for this is similar to the reasons above. Re-estimating stories changes the inputs to the backlog and throws off the numbers. Consistency is far more important than accuracy. Velocity will sort it all out.

Development

No change may go into master until it has undergone peer review in Stash and is successfully building in Bamboo. Master is to remain deployable at all times, so ensure it contains production-ready code with green builds.

Occasionally, due to intermittent problems with test timing or execution order, our master build will fail. Fixing the issue and ensuring it does not happen again becomes the highest priority when this happens.

Day-to-day development proceeds as follows:

1. The developer chooses a JIRA issue from among those in the active sprint, assigns it to himself, and moves it to the"In Progress" column of the agile board 2. The developer implements the change in a new git branch named after the issue number, e.g. EDSC-123. Commits are also prefixed with the issue number, e.g. "EDSC-123: Add user accounts page" 3. (Optional) When in need of collaboration or input, the developer pushes the branch and opens a new pull request without assigning a reviewer. Collaborators may pull the branch or view and comment on the diff in Stash. 4. When the change is complete, the developer pushes the branch, triggering a Bamboo build for that branch, and ensures the build is green. 5. Once the build is green, the developer assigns a pull request for the branch to another developer and moves the issue into the "Pending Review" column in JIRA. 6. The reviewer looks at the code, ensuring passing tests, adequate test coverage, code quality, and correct/complete implementation. He also runs the code and manually tests it. 7. The original developer fixes items brought up during review until the reviewer is satisfied. 8. Once satisfied, the reviewer merges the pull request, deleting the remote branch. 9. The developer moves the JIRA issue into the "Done" column, typically with a resolution of "Fixed." 10. Once deployed to the UAT environment, a primary stakeholder for the issue (for instance, the person requesting the change) tests the implementation and marks the feature "Closed" if satisfied or reopens it for additional work.

Technologies

Code structure

Our code generally follows Ruby on Rails conventions. The descriptions below describe useful conventions not outlined by Rails; they touch on the most important pieces of code and do not attempt to describe every directory.

- app/
  - assets/
    - javascripts/
      - config.js.coffee.erb The only place we keep per-environment configuration
      - util/ Domain-agnostic utility methods which may be generally useful for any Javascript application
      - modules/ Non-knockout components or customizations developed for Earthdata Search
        - maps/ Leaflet.js customizations
        - timeline/ Granule timeline implementation

      models/ Knockout.js models, cusomizations, and base classes
        - data/ Models for data items such as datasets and granules, separate from UI concerns
        - ui/ Models for UI elements, containing click handlers and tracking UI state
        - page/ Models for aggregating the UI and data models into a complete page state

    services/ Interfaces with 3rd party services
- lib/
  - echo/ client code for interfacing with ECHO, meant to be a thin wrapper able to be separated as a gem
  - vcr/ customizations to the VCR fixture library which configure fixture files, improve performance, and limit the number of merge conflicts we deal with in our fixtures.
- spec/ Test suite
  - features/ Capybara specs
  - javascripts/ Jasmine specs
- doc/ui/ UI documentation (pattern portfolio)

Server-side

Earthdata Search is implemented primarily using Ruby on Rails 4.1 running on Ruby 2.1 (MRI). For specific bugfix or patch releases, see the project's .ruby-version and Gemfile.lock files.

Production instances run on unicorn / nginx and are backed by a Postgres database.

Earthdata Search is primarily a client to numerous web-facing services:

- ECHO and the CMR: Metadata search, browse imagery, and data ordering
- DAAC-hosted OPeNDAP: Data acquisition and subsetting
- GIBS: Tiled imagery for granule visualizations
- URS: User authentication
- geonames.org: Placename completion
- ogre.adc4gis.com: Shapefile to GeoJSON conversion (to be replaced by self-hosted instance once we can stand up a node.js server)

Client-side

Earthdata Search uses a responsive HTML5 boilerplate from http://www.initializr.com for our basic layout. It provides default CSS "reset" rules, browser detection, and feature implementation for older browsers.

Client-side code is written in Coffeescript, and uses jQuery or plain Javascript for DOM interaction.

We use knockout.js for handling data models and keeping the interface in sync with changing data. Our knockout code can be found under app/assets/javascripts/models and is further subdivided into three types:

- /data: Models for data items such as datasets and granules, separate from UI concerns
- /ui: Models for UI elements, containing click handlers and tracking UI state
- /page: Models for aggregating the UI and data models into a complete page state

We use a heavily-customized Leaflet.js to draw our maps and Leaflet.draw to allow the user to draw and edit spatial bounds. Our key customizations are handlers for projection switches, rendering of GIBS-based layers for a set of granule results, and translation / interpolation of ECHO polygons into Leaflet- compatible geometries. Customizations are found in the app/assets/javascripts/modules/maps directory

For icon fonts, we use Font Awesome when appropriate and create custom icons through IcoMoon when Font Awesome does not suit our needs. IcoMoon files can be imported from the app/assets/fonts directory and customizations should be exported there.

Our granule timeline is a custom SVG component implemented in the app/assets/javascripts/modules/timeline folder.

We use a custom version of Bootstrap for some UI elements. Our customized version of bootstrap can be downloaded here.

We use a jQuery plugin for our datetime pickers. Usage information can be found here.

For the temporal recurring year range we are using bootstrap- slider

Testing

Fast and consistent tests are critical. The full suite should run in under 10 minutes and faster is better. If the suite gets slow, fix it. If a spec fails intermittently, find the problem and make it pass consistently. In order to ensure speed and consistency, we have mocked all of our external service calls using VCR and customized Capybara to avoid reloading sessions between every spec (generally this means you want to use `before(:all)` instead of `before(:each)` in specs and ensure that there is a corresponding `after(:all)` block that resets the page state.

The entire suite of Earthdata Search tests, including unit, functional, and UI tests, may be run using the `rake test` command.

Earthdata Search uses RSpec for its unit, functional, and integration tests. Tests may be run by using the `rspec` command in the project root.

Integration specs use Capybara and CapybaraWebkit to simulate browser interactions. We include the capybara-screenshot gem and publish screenshots produced by failing builds to aid debugging.

We document the application's behavior using our RSpec integration specs. To generate this documentation, run `rspec spec/features/ --format=documentation -o doc/specs.html`. Generated documentation will appear in doc/specs.html.

In order to allow us to describe the application behavior using RSpec, developers must read and follow the guidelines in the "Testing" section of this document's style guide.

For testing Javascript, we use Jasmine, which has an RSpec-like syntax. Developers should exercise their Javascript in the same way they exercise Ruby code. Jasmine tests are located under spec/javascripts. They can be run continuously using `rake jasmine` or once using `rake jasmine:ci`.

The Pattern Portfolio for HTML, CSS, and Javascript testing

In addition to the RSpec and Jasmine tests documented in the previous section, we perform additional tests to ensure that our UI looks and functions as intended.

Reusable CSS rules and Javascript components should be displayed in the project's pattern portfolio document found at docs/ui/index.html. Developers may add to the portfolio by editing docs/ui/templates/index.html.erb. We generate the portfolio by running `rake doc:ui` in the project root (or `rake doc` to generate all project documentation).

To see which rules are demonstrated, we run Deadweight on the pattern portfolio to scan for unused rules using `rake test:deadweight`.

To ensure the quality of our CSS, we run CSS lint using `rake test:csslint`. Developers are encouraged to read the CSS Lint wiki (https://github.com/stubbornella/csslint/wiki/Rules) to learn about the reasoning behind the rules.

Deployment

Local Environment

We run locally using Pow in a server called edsc.dev. We use sqlite databases in local test and development environments.

Shared Environments

There are 3 shared deployment environments for Earthdata projects, including Earthdata Search:

System Integration Testing (SIT)

```
An environment for performing integration tests on builds, ensuring that new code works correctly in a
production environment before placing it in front of customers or users. This is roughly equivalent to
ECHO's "testbed" environment.
```

User Acceptance Testing (UAT)

```
An environment for verifying that builds meet the customer and user expectations before promoting them to
operations. We expect partners and advanced users to use this environment frequently for their own testing,
so it will be public and must have a high uptime. This is roughly equivalent to ECHO's "partner-test"
environment.
```

Operations (Ops)

```
The production environment, running the canonical public-facing site.
```

Deploying changes

Changes to shared environments must be deployed through Bamboo via the Earthdata Search deployment project.

Development Style Guide

Introduction and Concepts

The Pattern Portfolio

See "The Pattern Portfolio for HTML, CSS, and Javascript testing"

Progressive Enhancement

Progressive enhancement involves starting with a baseline HTML document that is readable and usable without any styling or javascript. We accomplish this by using semantic markup. From there we enhance the markup by unobtrusively adding CSS styles and Javascript.

By starting with working basic HTML, we ensure we have a page that's minimally usable by:

- Visually impaired users relying on screen readers
- Web crawlers which do not use CSS to determine content's importance and possibly do not evaluate Javascript
- Users who experience an error that breaks scripts or the serving of assets
- Test suites which otherwise do not need Javascript or CSS to perform their tests
- Users of unsupported browsers
- Users with Javascript or CSS disabled or who are attempting to access data from scripts (common in the scientific community)

The key point here is that a missing browser feature or a single script or style error should not render the site unusable.

Mobile-first Development

This is similar to Progressive Enhancement described above. We target narrow screens (typically mobile screens) first and add additional styles as screens get wider, using media selectors.

The reason for this is mostly practical. Mobile renderings tend to have a much simpler set of styles than larger renderings. If we targetted large screens first, much of our mobile effort would be in undoing the styles aimed at larger screens.

Another key point here is that we will plan to be mobile friendly from the start. It is much easier to build this in from the beginning than to attempt to construct it later on.

General Principles

Treat frontend authoring as a development discipline.

HTML is just markup and easy to learn. CSS is turing complete, but not really a programming language. They're easy to dismiss.

The reality is, though, that HTML and CSS provide very few mechanisms for code reuse and organization. Their size and complexity has a direct and perceptible impact on page speed, and they are the most user-visible part of the codebase. It is exceedingly difficult to create clean, performant, reusable, and extensible frontend assets. It generally requires much more care than the corresponding backend code, since backend languages are designed with these aspects in mind.

Frontend authoring is a development discipline and requires a great deal of care and consideration, to the point that most of this guide focuses on frontend development.

Minimize complexity from the beginning.

It is typically very difficult to extract complexity from front-end code. All new components should be focused on reuse, versitility, and extensibility. When possible, do not add new components at all, but reuse or extend existing components, which can be found in the Pattern Portfolio.

Create components, not pages.

When developing frontend code, the unit of reuse should be the module, component, or behavior, not the page. Design components that can be used across multiple pages or that are versatile enough to be used multiple times on the same page, possibly for different purposes. Write, CSS, Javascript, and partial HTML for components, not for pages, in order to promote robustness and reuse and keep code size in check.

Document decisions.

There may be good exceptions to every rule in this guide. When in doubt, follow the guide, but make exceptions as necessary. Always document these decisions. Further, whenever you write code in a non-standard way, or you are faced with multiple competing options and make an important choice, document those decisions as well.

Testing

Test everything.

Every line of application code, including UI markup, should be exercised in a test.

Test at the appropriate level.

Exercise boundary conditions, error handling, and varying inputs at the unit or functional level.

Integration specs should demonstrate user-visible system behavior.

Remember that integration tests run much more slowly than unit tests, so prefer to test more thoroughly at the unit level.

Integration tests should be placed in the "spec/features/" folder. All other tests should go in the default locations generated by Rails (e.g. "spec/models/")

Build meaningful sentences with RSpec blocks.

The chain of RSpec "describe" and "context" blocks leading up to the final "it" block should form a human-readable sentence. This is particularly true for integration specs where we are documenting system behavior spec names.

Consider an example where we don't use this style.

Bad Example:

```
describe "Account creation" do
      …
      context "messages" do
        …
        it "should display success messages" { … }
        it "should display failure messages" { … }
      end
      it "recovers passwords" { … }
      it "should send emails to users" { … }
    end
```

Consider the sentences produced by the above:

1. Account creation messages should display success messages. 2. Account creation messages should display failure messages. 3. Account creation recovers passwords. 4. Account creation should send emails to users.

The spec fails to describe the system. Reading the sentences, we don't know why a particular behavior might happen. Some of the sentences don't entirely make sense.

We fix the problem by using more descriptive contexts and paying attention to the sentences we're constructing with our specs.

Improved Example:

```
describe "Account creation" do
      …
      context "for users providing valid information" do
        it "displays a success message" { … }
        it "sends an email to the user" { … }
      end
      context "for users providing duplicate user names" do
        it "displays an informative error message" { … }
        it "prompts users to recover their passwords" { … }
      end
    end
```

Consider the sentences produced by the above:

1. Account creation for users providing valid information displays a success message. 2. Account creation for users providing valid information sends an email to the user. 3. Account creation for users providing duplicate user names displays an informative error message. 4. Account creation for users providing duplicate user names prompts users to recover their passwords.

The above sentences more adequately describe the behavior of the system given varying inputs.

Avoid the ugly mirror problem.

http://jasonrudolph.com/blog/2008/07/30/testing-anti-patterns-the-ugly- mirror/

Tests should describe how the system responds to certain inputs. They should not simply duplicate the code under test.

Avoid over-mocking.

Ruby makes it very easy to stub methods and specify return values. Often, this can lead to fragile tests which don't perform any useful validation. If a test stubs every call made by a method, for instance, the test doesn't verify that the method actually works; simultaneously, the test will break any time the method changes.

Mocks couple tests and code, and should be used very sparingly. Valid reasons to use mocks include:

1. Calls to external services which cannot or should not be made during tests. 2. Calls which are expensive to perform (I/O) and irrelevant to the test at hand. 3. Calls which would require an unreasonable amount of setup and fixtures and are irrelevant to the test at hand.

When in doubt, it's better to not mock.

Minimize test suite execution time.

The test suite should provide developers with rapid feedback regarding the correctness of their code. To accomplish this, they should execute quickly. Keep performance in mind when writing tests. The following guidelines will help minimize execution time:

1. Test varying inputs and edge cases at the unit or functional level, rather than the integration level. 2. Avoid running integration tests with Javascript enabled unless Javascript is necessary for the feature under test. 3. Avoid calling external services, particularly ones which cannot be run in a local environment. Use mocks for these services. 4. Avoid loading external images, CSS, and Javascript in integration tests. 5. Avoid or disable interactions and interface elements that will cause Capybara to wait. For instance, disable animations or transitions. 6. Skip to the page under test in integration tests, there is no need to start at the home page for every spec (though you should have a spec which verifies you can start at the home page). 7. Avoid increasing timeouts to fix intermittent problems. Find other means. 8. Time your tests. 9. Follow this style guide for performant HTML, CSS, and Javascript.

If performance becomes a problem, we may segregate tests into "fast" and "full" runs, but ideally we will avoid this.

Fix sources of intermittent failures immediately.

If you see a failure and you suspect it was caused by some intermittent problem, e.g. a timeout that is too short or an external service being down, it is not enough to simply re-run the tests. Fix the problem. If a problem truly cannot be fixed, document why, catch the specific error that cannot be fixed, and throw a more meaningful one.

Exercise all CSS, HTML components, Rails partials, and Rails helpers in

the Pattern Portfolio

Factor non-trivial HTML into partials and helpers and demonstrate its use in the Pattern Portfolio. This allows other developers to find and reuse partials and ensure that new code does not break existing code.

Display Javascript components in the Pattern Portfolio

Catalog custom Javascript components in the Pattern Portfolio. Show examples of configuration options, if appropriate.

Test Javascript components using Jasmine in offline pages

Create offline pages, similar to the Pattern Portfolio, to exercise Javascript components in their various states. Avoid requiring a running application instance or performing server communication in Jasmine specs.

Integration specs should still perform simple checks on Javascript components, but the bulk of Javascript testing should be performed offline.

HTML and ERB

Demonstrate markup patterns in the Pattern Portfolio.

Factor any non-trivial markup patterns into partials or helpers. Non-trivial markup patterns include patterns which require an element to have a specific child or children to obtain a certain behavior or style. For instance, a header containing a span child may trigger CSS image replacement behavior.

Demonstrate all markup in the Pattern Portfolio. Call helpers and partials within the portfolio to ensure that markup stays in sync.

Check the pattern portfolio before adding new components.

It is better to reuse, extend, or modify an existing component than to create a new one. In all cases, we want to minimize the complexity of the code and markup we send to the browser.

Use semantically meaningful elements where possible.

`div` and `span` convey no meaning and are rarely the most appropriate element in a given context. When adding an element, look at other elements that HTML provides to see if another is more appropriate. This helps developer readability, keeps styles simpler, and may aid accessibility or search engine indexing. Here are some questions you may ask:

- Does it contain a paragraph? Use a `p`.
- Does it contain a heading or subheading? Use a `h[n]`.
- Does it contain an item in a list of items? Use a `li`.
- Does it constitute a section of the page appropriate for an outline? Use a `section`.
- Does it contain header or footer information for a page or section? Use a `header` or `header`.
- Does it contain describe a form field? Use a `label`.
- etc

Avoid presentational class names.

We should be able to dramatically alter the look and feel of the site without changing (much) HTML in order to match changing designs or to accommodate new user agents. This is particularly important for responsive design, where multiple browsers may need to render the same HTML in different ways.

To allow this, use class names that describe an element's purpose, contents, or behaviors, rather than ones that describe its presentation.

```
Overly-presentational class names.  We may not want this element
      to appear to the left on mobile browsers, for example.
    -->
    <section class="left bold bordered">

    <!-- This one uses more semantically appropriate class names -->
    <section class="primary emphasized summary">
```

Avoid unnecessary elements. Keep nesting shallow.

Minimize the overall depth of HTML to decrease page size, increase readability, and improve rendering speed.

Avoid conditional logic in views for partial display

Use view helpers for arbitrary display of content

See This example and description from Stack Overflow. Use Rails helpers to dynamically generate highlight dynamic HTML with widely varying elements and structure. Use Rails partials to generate more static content. Remember to demonstrate partials and helpers in the pattern portfolio.

CSS with Sass

Earthdata Search uses SCSS to generate its CSS. It follows the guidelines for scalable CSS outlined by SMACSS, with key items reproduced in this document. The CI build checks CSS style using CSS Lint. Developers are strongly encouraged to read the CSS Lint rules.

Exercise every line of CSS in the pattern portfolio.

Use the Pattern Portfolio to demonstrate all styles in the application. This allows other developers to find styles that already exist instead of re-inventing the wheel, and it allows developers to ensure that new styles do not break existing styles. The CI build ensures that each style is used at least once in the portfolio.

Do not use id selectors (#id)

ID selectors indicate that a style is one-off and can only appear in one place on a page. Prefer to build reusable styles that could appear in multiple places or on multiple pages. Further, id selectors are very specific in terms of the CSS cascade, making them hard to override.

There are two exceptions to this rule: #site-header and #site-footer. These two areas of the page are typically significantly different than the rest of the site to the point where they need to override all styles, and the drawbacks of using id selectors mostly do not apply.

Do not use style attributes or inline styles.

Place all styles in CSS files. Mixing styles with HTML makes tracking, testing, and reusing styles much more difficult. Further, style attributes are incredibly specific and difficult to override.

Javascript components should also prefer to apply classes to elements rather than alter their style attributes for the same reasons.

Do not use `!important`

`!important` makes styles impossible to override unless the overriding rule also uses `!important`. In almost every case, using `!important` is unnecessary. Prefer to use selectors with higher precedence or alter overridden selectors to have lower precedence.

Very very rarely it is necessary to use `!important` because of third-party code. This typically happens when a third-party Javascript library adds an undesirable style attribute to an element. When given the choice between altering third-party Javascript or using `!important`, the latter is usually preferable. In this circumstance, document the decision. This is one reason we avoid setting style attributes, even in Javascript components.

Prefer selectors that describe attributes and components, not domain

concepts.

Consider the following element:

```
<ul class="granules">
```

`ul.granules` selects a list with very specific elements, which may only be available on one page. Any styles applied to this list are unlikely to be reusable. Adding classes that describe attributes of the list makes CSS styles more modular and reusable.

```
<ul class="granules zebra scrollable selectable">
```

Here we may use different selectors to apply different attributes to the list. `ul.zebra` may add zebra striping to the rows, `ul.scrollable` may add a scroll widget, and `ul.selectable` may provide hover and selection mechanisms. Any of these attributes could be useful on lists that do not describe granules, and other lists could mix and match attributes to match their needs. Fine-grained attribute-centric class names provide for better reuse.

Follow SMACSS rule categories and naming conventions

Follow SMACSS guidelines for grouping and naming CSS classes. Prefix layout classes with "l-". Prefix state classes with "is-". Do not prefix module classes. Use double dashes to indicate module subclasses, e.g. "module--subclass"

Minimize selector depth

As described by SMACSS' Depth of Applicability chapter, minimize and simplify CSS selectors. Deeply nested selectors are easy to create in Sass, but they are hard to understand and create a strong coupling to the underlying HTML structure. Further, they tend to be overly-specific, causing duplication in CSS rules. Whenever possible, keep nesting 1-2 selectors deep, with 3 being the maximum.

```
// Deep nesting
    .module {
      div {
        h1 {
          span {
            font-weight:bold;
          }
        }
      }
    }
    // Shallower nesting
    .module {
      h1 > span {
        font-weight:bold;
      }
    }
    // Shallow nesting
    .module-header-text {
      font-weight:bold;
    }
```

Use fast CSS selectors where possible.

A large page with many CSS rules can suffer rendering performance problems that make pages feel sluggish even on modern browsers and computers. Understand selector performance and follow these rules to allow pages to render quickly:

1. Use child selectors 2. Avoid tag selectors for common elements (div, span, p) 3. Use class names as the right-most selector

Use variables for colors and numbers.

Use Sass variables to describe all colors except black and white and all numbers except 0 and 1px. This makes it easier to find usages of measurements and change them as necessary.

Use Sass helpers for CSS3 styles.

Sass and compass provide helpers for CSS3 styles that normalize experimental browser extensions, for instance

```
@include border-radius(4px, 4px);
```

generates

```
-webkit-border-radius: 4px 4px;
    -moz-border-radius: 4px / 4px;
    -khtml-border-radius: 4px / 4px;
    border-radius: 4px / 4px;
```

Use these helpers to avoid overly-verbose CSS.

Target browsers with classes not CSS hacks.

The base site contains HTML boilerplate libraries which add CSS classes to the html and body element that detect commonly misbehaving browsers (older IE versions) and browser capabilities. Use these classes to target browsers or capabilities rather than relying on CSS hacks.

Javascript (Coffeescript)

Use Coffeescript

New project code should be written in Coffeescript. It eliminates much of the boilerplate and gotchas of Javascript, producing easier-to-read code that is more accessible to developers with all levels of front-end experience.

Follow polarmobile's Coffeescript style guide for code formatting

https://github.com/polarmobile/coffeescript-style-guide

Build components and modules, not one-off elements

Try to build Javascript components and widgets that could apply throughout the site, rather than on a single page or in a single situation. Good questions to ask when writing code is "Can I make this into a widget?" or "Can I apply this behavior to all elements with this class?". If the answer is no, perhaps the element could be described as a composition of multiple components (scrollable, zebra-striped, selectable list rather than one-off granule list)

Allow users to bookmark content and use the back button

Dynamic interfaces are great, but users should be able to bookmark their current location to return later, especially for complex search UIs like Earthdata Search. Further, we should allow the user to back up to previous states or leave the site via the back button.

When building the interface, use the History API to ensure that history entries are pushed to the stack appropriately. Push entries to the stack when the user reaches points they would reasonably expect to bookmark. Avoid pushing entries so frequently that backing out of a state using the back button becomes tedious or impossible.